LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-446

MULTIPROCESSOR ADDRESS TRACING AND PERFORMANCE ANALYSIS

Anant Agarwal
David Chaiken
David Kranz

April 1991

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM 446	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-87-K-0825
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Multiprocessor Address Tracing and Performance Analysis

12. PERSONAL AUTHOR(S)
Agarwal, A., Chaiken, D., Kranz, D.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	April 1991	22

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The design of multiprocessors requires using data from real parallel programs to study the cost-performance tradeoffs in memory systems and interconnection networks. Practical methods of multiprocessor performance analysis use address traces to encapsulate the behavior of parallel programs. Trace-driven simulation techniques bridge the gap between full system simulations and analytical models; they are faster than full system simulations, and they can yield better predictions than analytical models. This paper reviews various multiprocessor address tracing schemes and presents the design and performance of a compiler-aided tracing scheme that efficiently obtains traces of systems with an arbitrary number of processors. The paper describes coupled and decoupled classes of trace-driven simulation schemes, and compares their relative speed and accuracy. Decoupled techniques simulate the various components of the multiprocessor separately, without modeling the feedback between the processor, the memory system, and the interconnection network. Coupled techniques simulate the feedback between multiprocessor components. We validate a decoupled trace-driven simulation scheme that is over an order of magnitude faster than coupled schemes, yet has comparable accuracy.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Carol Nicolora	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

*U.S. Government Printing Office: 1986-607-047

Unclassified

Multiprocessor Address Tracing and Performance Analysis

David Kranz, David Chaiken, and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

The design of multiprocessors requires using data from real parallel programs to study the cost-performance tradeoffs in memory systems and interconnection networks. Practical methods of multiprocessor performance analysis use address traces to encapsulate the behavior of parallel programs. Trace-driven simulation techniques bridge the gap between full system simulations and analytical models; they are faster than full system simulations, and they can yield better predictions than analytical models. This paper reviews various multiprocessor address tracing schemes and presents the design and performance of a compiler-aided tracing scheme that efficiently obtains traces of systems with an arbitrary number of processors. The paper describes coupled and decoupled classes of trace-driven simulation schemes, and compares their relative speed and accuracy. Decoupled techniques simulate the various components of the multiprocessor separately, without modeling the feedback between the processor, the memory system, and the interconnection network. Coupled techniques simulate the feedback between multiprocessor components. We validate a decoupled trace-driven simulation scheme that is over an order of magnitude faster than coupled schemes, yet has comparable accuracy.

1 Introduction

The performance of large-scale multiprocessors hinges critically on the efficiency of their memory systems and interconnection networks. In turn, the behavior of memory systems and networks depends both on their architecture and on the memory referencing nature of parallel programs. Consequently, the design of multiprocessors requires the study of cost-performance tradeoffs in memory and interconnection network architectures using data from real parallel programs. If a benchmark suite of parallel processor address traces were available, it would be feasible to make design decisions based on quantitative results rather than seat-of-the-pants analysis.

One method of quantitative multiprocessor performance evaluation is based on trace-driven simulation. Trace-driven analysis of multiprocessors involves simulating a software model of a system, using parallel address traces as the stimulus. Such simulations are typically used to evaluate multiprocessor performance, taking into account memory system behavior and interconnection network effects. Parallel address traces find a natural use in multiprocessor cache and memory system analysis. Multiprocessor cache simulations provide statistics such as cache miss rates and interconnection network traffic patterns. Simulations can also generate network

request traces that record the transactions requiring network traffic, such as cache misses and coherence-related invalidations and acknowledgments.

Predicting overall multiprocessor performance requires the combination of cache and memory performance statistics with the effects of the interconnection network. This paper describes two classes of simulation techniques that combine memory and network effects to obtain overall multiprocessor performance. *Decoupled* techniques use address traces to analyze the cache and the network components separately, without simulating their interactions. *Coupled* approaches, on the other hand, incorporate feedback from the network into the cache analysis. Coupled methods use synchronization information embedded in address traces to integrate the trace generation mechanism into the simulation system.

Without trace data what are the choices available to the designer? Analytical models are one approach. Unfortunately, theoretical analyses are only as good as the parameters used to drive them. These parameters commonly represent workload characteristics. Without traces, common forms of parameter estimation use guesswork or intuition. Sometimes parameters can be measured from small scale systems that currently exist, but the deficiency of this method is apparent: without knowing application characteristics, the models might make unrealistic assumptions.

Alternatively, full system simulation can be used to evaluate multiprocessor performance. Because this scheme most closely matches a real machine implementation, it is highly accurate. However, its close resemblance to a real implementation implies that building such a simulator suffers from many of the difficulties of building a real machine. Furthermore, a detailed software simulation of a complete machine with its operating system is inflexible and slow. For example, we have implemented a full system simulator called ASIM [6] to aid the design of Alewife, a large-scale multiprocessor being built at MIT. It is now possible to compile a parallel program, to link the program with a runtime system that dynamically partitions and schedules tasks, and to run the program on a simulated version of the Alewife architecture.

Due to the complexity of ASIM, implementing and running full system simulations requires an order of magnitude more time than our decoupled trace-driven simulations. Despite the overhead of full system simulation, we have discovered that it is extremely valuable in the final phases of machine design. Once the architecture of the machine has solidified enough that flexibility is not of great concern, a full simulator serves as a tool for architectural verification, test vector generation, and software development. However in the primary stages of development, trace-driven simulations coupled with analytical models can help evaluate a larger number of multiprocessor design alternatives rapidly and accurately.

The trace-driven analysis approach bridges the gap between full system simulation and analytical modeling. Traces can function as an intermediate form for containing dynamic parallel program characteristics. The use of traces can eliminate the need to write a detailed processor simulator and an operating system, even in a coupled simulation environment. Although traces can have diverse antecedents in terms of the language, the compiler, and the processor used in their creation, a canonical format makes them a convenient machine-independent representation of parallel program behavior. When traces are made generally available, they allow researchers to draw upon data from a large class of applications written in various languages. The overhead of writing test applications can thus be shared by a large community.

A criticism often leveled at trace-driven evaluation asserts that such analysis leads the de-

signer to repeat the mistakes of past designs. Such may indeed be the case if traces are strongly tainted with architecture specific characteristics. However, a good tracing scheme captures traces that reflect applications' intrinsic characteristics and programming models, rather than the behavior exhibited by a particular machine.

Traces have been invaluable in single processor system design and in the design of small-scale multiprocessors. Traces and tracing methods can be even more useful in large-scale multiprocessor designs. Typical applications of traces include the analysis of interconnection networks, synchronization, scalable cache coherence schemes, and multithreaded processors. Tracing also provides dynamic execution profiles of parallel programs when such profiles are hard (if not impossible) to get by examining application code. For programs that exhibit memory access patterns that are statically predictable, one might argue that traces do not yield much insight. This is true, but such applications are easily analyzed and optimized by compilers or programmers. The hard class of applications to analyze or to tune are those whose dynamic memory access patterns are not predictable at compile time.

In this paper, we survey several multiprocessor tracing schemes developed in the past and describe the design of Tmul-T, a compiler-aided tracing scheme. Tmul-T yields accurate address trace data for an arbitrary number of processors and traces the runtime system of Mul-T as well. An implementation of Tmul-T on the Encore Multimax suffers a slowdown of less than a factor of 20 over that of an uninstrumented processor. The discussion of multiprocessor address tracing methods is followed by trace-driven simulation techniques for multiprocessor performance evaluation. We compare coupled and decoupled schemes that use various combinations of simulation and analytical modeling. The results from coupled simulations validate an efficient form of decoupled trace-driven analysis that combines simulation and analytical modeling.

The rest of this paper reviews several existing tracing systems in Section 2 and then presents the compiler-aided Tmul-T scheme in Section 3. Section 4 compares several evaluation techniques that use address traces to evaluate the performance of multiprocessor memory systems, and Section 5 summarizes our experiences with trace-based multiprocessor evaluation.

2 Multiprocessor Tracing Schemes

In the recent past, a number of schemes have been developed to yield parallel address traces. This section reviews several general techniques: hardware tracing, microcode aided tracing, tracing using a trap bit, and post-mortem scheduling from a single processor trace. Compiler-aided tracing schemes are discussed in the next section. We evaluate the various schemes based on how well they meet our goals for multiprocessor tracing, namely:

1. The generated traces should represent a possible real execution.
2. The traces should not be affected by the peculiarities of the machine being used to generate the data.
3. The traces should contain information about task behavior and synchronization, as well as raw address references.
4. The scheme should be able to generate large traces of an arbitrary number of processors.

5. Tracing should not reduce the speed of the machine inordinately.

2.1 Hardware Tracing

Hardware tracing schemes typically observe addresses on a system bus and store them into a buffer. If each processor has its own buffer, then some mechanism must exist to re-create the global temporal ordering of addresses. Hardware tracing has been implemented on several multiprocessors, including the Sequent and the HP Spectrum-based machines. The major problems with this method include trace-length limitations due to the physical size of the buffer memory, and a limit on the number of processors traced. In addition, if addresses are traced after filtering by a cache, then fine-grain effects are hard to capture.

2.2 Microcode Tracing

ATUM, a microcode-based scheme for multiprocessor tracing is presented in [20]. In this scheme, the microcode of a machine is modified to record the address of a memory reference into a portion of main memory reserved to hold traces. Because the traces are stored in main memory, the trace length is limited only by physical memory size. This scheme has yielded several traces that are several million references long. The major limitation of the above implementation of microcode-based tracing is that only two to four processors are traced, because the host machine does not support more processors. Both hardware tracing and microcode-based tracing yield complete user and system references; however, the microcode tracing scheme slows processor execution by a factor of twenty.

One of the extensions to the basic microcode scheme proposed in [3] performs simulations or event counting in microcode. Such a method has the advantage of allowing simulations of arbitrary length, while avoiding trace storage problems. ATUM implements a cache in microcode to filter out repeat instruction words. The ATUM implementation also records instructions, process identifiers, processor numbers, and physical to virtual translations in the trace in order to observe data sharing.

2.3 Trap-Bit Tracing

Several implementations of single processor tracing have made use of a *trap bit* [11, 19]. In these schemes, the processor interrupts user code if the trap bit is set – a feature commonly used by debuggers. For tracing, the trap code simply interprets the instruction that trapped and records the memory address. Arturo Salz's VTRACE [19] also includes an optimization that caches some of the previous instruction interpretations in a software cache, for later reuse. Steve Goldschmidt modified the scheme at Stanford to yield multiprocessor traces. Kai Li and Leslie Matheson have developed a similar tracing scheme at Princeton.

The multiprocessor T-bit tracing technique schedules a new process on every trap instruction. After a process traps and the corresponding memory address is recorded by the trap code, the scheduler saves the processor state of the trapped process. It then schedules another process from its list of processes, typically in a round-robin fashion. The execution of the parallel program

by the above mechanism is a valid execution because synchronizations are obeyed in the normal manner.

The major benefits of this method are that an arbitrary number of processes can be traced, and that tracing can be achieved on a single processor. The Stanford implementation runs on any VAX machine that runs the MACH operating system. Because of the need to trap the processor on every memory reference the scheme is very slow, typically yielding about 100 addresses per second on a VAX 8350 composed of ~1 MIPS processors. There is virtually no limit on trace length, because the traces can be stored on disk. Sharing is detected by writing out the limits of the shared memory region provided by MACH. This scheme can be made much faster by using threads in the implementation of processes. Unfortunately, this tracing mechanism cannot handle the dynamic creation and destruction of tasks. The scheduler keeps track of a fixed number of processes (or virtual processors) that are spawned at the start of execution, and can not change the number in the middle of the trace.

The major problem with all the above approaches lies in the fact that while the order of memory references is correct for the execution that produces a trace, the order may not be consistently maintained when a different architecture is simulated. Some analyses using the traces do not cause a change in the relative order of addresses, for example, a coherent cache simulation measuring the invalidation rate assuming an infinitely fast network. Other analyses, such as a coherent cache simulation using a realistic network, will surely distort results. Section 4 describes this ordering problem in more detail and suggests several solutions.

2.4 Post-Mortem Scheduling

The post-mortem scheduling technique uses the trace of a uniprocessor execution of a parallel application to generate a multiprocessor trace. The uniprocessor trace is a task trace with embedded synchronization information that can be scheduled after execution (post-mortem) into a parallel trace obeying correct synchronization constraints. Of course, the method assumes that a trace of a single processor with explicit synchronization markers can be generated. This methodology uses only one processor to generate the trace and to schedule it afterwards. The number of processes is limited only by synchronization constraints and the number of parallel tasks in the single processor trace. We now describe an implementation of this method by Cherian with So at IBM [7] that we use for the analysis in Section 4.

A uniprocessor execution of an application parallelized using the single-processor-multiple-data (SPMD) computational model yields a single processor trace. Single processor traces are gathered using PSIMUL [21], a system for tracing parallel applications on IBM S/370 machines. The information included in the uniprocessor execution trace is the key to this scheme. In the SPMD model, each code section (task) in the system starts and ends with a synchronization event. A single processor trace is a description of the memory reference behavior of all the tasks with temporal ordering constraints, but without an actual schedule of events.

The post-mortem scheduler produces a parallel trace with the required temporal ordering by simulating processors executing the parallel application described by the uniprocessor trace. The scheduler first makes a pass through the uniprocessor trace and constructs a task trace from the synchronization markers. The scheduler then simulates the processors executing these tasks in a round-robin fashion with each processor making one reference each cycle from its task. A

useful memory-conserving method in the scheduler uses direct-access I/O to maintain multiple simultaneous links into the different task segments of the single processor trace file.

The scheduler also simulates the synchronization behavior of the processor and outputs appropriate synchronization references into the multiprocessor trace. In Cherian's implementation, the scheduler uses busy-waiting for the synchronization model. Kurihara has modified Cherian's implementation to include other models of barrier synchronization such as adaptive backoff and distributed barrier trees [17]. Kurihara has also extended this implementation to allow coupled simulations by incorporating feedback from a memory system simulator. In this coupled post-mortem scheme, a processor issues a memory request from its thread only after its previous network request is satisfied. The processor can also choose to switch to a different process on a network request. Other researchers have developed similar schemes. Kumar and So also use barriers, and their method causes processors to busy wait at synchronization points [16]. Mark Holliday has proposed a method that dynamically schedules address trace segments using program constraint graphs [12]. The coupled and decoupled schemes are compared in Section 4.

3 Compiler-aided Tracing

A basic problem with the tracing methods reviewed thus far is that the system must perform a dynamic check to determine exactly when to write an address to the trace log. The dynamic check causes those methods to be slow. An alternative method uses the compiler to insert the logging code where necessary, modifying the object code.

There are two such compiler-aided tracing methods. In one, the compiler simply inserts the logging instructions and the program is run on an extant multiprocessor as in [9] and [22]. Our scheme, called Tmul-T, inserts logging instructions, but also provides for the *virtualization* of the multiprocessor.

3.1 Virtualization

The key to our method is the notion of *virtual processors*. The user program compiles into tasks that run on the virtual processors. At the same time, the virtual processors run on some number of real processors. This organization is shown in Figure 1. The virtual processors are scheduled on the real processors in a round-robin manner, with a new virtual processor running each time a memory reference occurs. This scheduling policy ensures that the program behavior represents a possible real execution of an untraced program, since the only communication between processors is through shared memory.

This method has the disadvantage of incurring the cost of unloading and loading a thread at every reference, unlike other compiler-aided tracing schemes. On the other hand, because our multiprocessor is virtualized, we can gather traces for an arbitrarily large number of processors. Furthermore, a minor extension of this scheme allows the *emulation* of multiprocessors with an arbitrary number of processors. In addition, we can isolate the traced data from certain characteristics of the machine generating the traces.

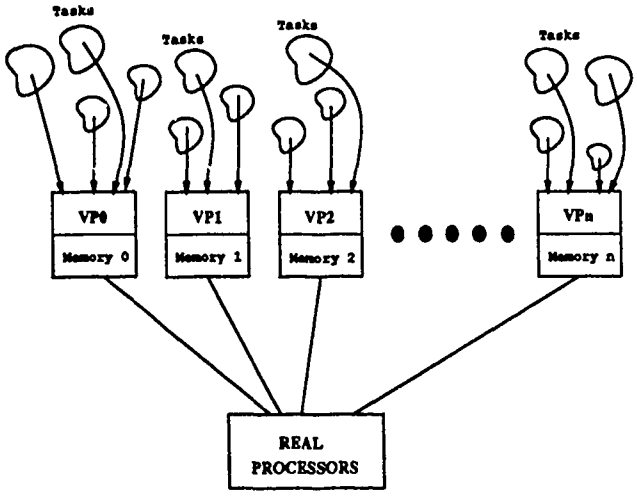

Figure 1: Tasks are scheduled on virtual processors VP0 to Vn, and virtual processor are scheduled on one or more real processors.

3.2 Tmul-T

Address tracing schemes for shared-memory multiprocessors represent tradeoffs between performance and faithful generation of address traces that would actually occur on a machine without tracing instrumentation. In this section, we demonstrate a tracing methodology called Tmul-T that meets the goals for tracing schemes specified at the beginning of Section 2.

Tmul-T (pronounced *tee mul tee*) is our variant of compiler-aided tracing based on Mul-T, a parallel Lisp system [13]. Tmul-T can be used to generate memory address traces for Mul-T applications or to emulate applications running on an arbitrary number of processors. In addition, it generates information about events in the program such as the creation of a task. (See Appendix B for a list of events traced.)

Let us first review the Mul-T system briefly. Mul-T is a parallel Lisp system that runs on an Encore Multimax multiprocessor. It is an extended version of the T system [18] that supports parallel processing using Multilisp's `future` construct [10]. Mul-T uses a modified version of T's ORBIT compiler [14] to generate native code for the Multimax's NS32332 processors.

Mul-T (like Multilisp) is an extended version of Scheme [1], a lexically scoped dialect of Lisp. Mul-T's execution environment contains the same sorts of data types and primitive operators as Scheme or any Lisp dialect. In Mul-T, however, many threads of computation, or *tasks*, can be active simultaneously, manipulating objects in a single shared heap.

Mul-T's basic mechanism for generating concurrent tasks is the `future` construct. The expression (`future` X), where X is an arbitrary expression, creates a task to evaluate X and also creates an object known as a *future* to eventually hold the value of X. When created, the future is in an *unresolved*, or *undetermined*, state. After the value of X becomes known, the future *resolves* to that value, effectively mutating into the value of X and losing its identity as a future. Concurrency arises because the expression (`future` X) returns the future as its value without waiting for the future to resolve. Thus, the computation containing (`future` X)

can proceed concurrently with the evaluation of X. When execution of a Mul-T program is not made explicitly parallel using **future**, it is sequential.

3.3 Implementation of Tmul-T

Compiled code running in Tmul-T has the same basic instructions as the compiled Mul-T code except that each instruction that references memory is preceded by an subroutine call to the Tmul-T kernel. The arguments to the subroutine call are generated by the compiler and consist of an opcode describing the kind of memory reference, *e.g.* read or write, and an address being referenced. This subroutine saves the state of the virtual processor and switches to the next processor in round-robin fashion. Thus each virtual processor runs the same code as would a real processor in Mul-T until a memory reference is made. We call this the *emulation* mode of Tmul-T. In *tracing* mode, the kernel subroutine writes a packet into a memory buffer before switching to the next virtual processor. These packets accumulate in memory and are periodically written out to the disk. Each packet contains three fields:

1. The processor field contains the identification number of the virtual processor making the memory reference. Due to the round-robin scheduling of virtual processors, this field is not strictly necessary, but it allows other kinds of scheduling to be used.
2. The opcode field contains a number indicating which type of memory reference is being made, e.g. read or write.
3. The address field contains the address being referenced.

A program running under the Tmul-T emulation system exhibits the same execution behavior as if it were uninstrumented. Synchronization is unchanged by Tmul-T, because if a processor would be spinning or blocked in Mul-T, the virtual processor in Tmul-T will also spin or block. Communication remains the same, because the tracing scheme slows all processes by approximately the same ratio. Since the virtual processors transmit information only through shared memory, Tmul-T captures the communication structure of the program by switching between virtual processors on every memory reference. This communication pattern can later be reconstructed from the information in the trace.

Tmul-T has many of the same properties as the T-bit tracing that we described earlier. However, Tmul-T runs faster because the compiler-aided scheme knows when to generate code to write an address packet into memory, while the T-bit method traps every instruction, whether or not it references memory. Since Tmul-T's virtual processors are implemented as *lightweight* tasks, the tracing overhead includes only the time needed to pass arguments to the internal procedure call and the time to save and to restore the virtual processor registers. In comparison, the T-bit method incurs both an expensive heavyweight process switch and the overhead of interpreting the trapped instruction to determine the associated memory address.

In addition to recording raw memory access data, Tmul-T produces information about how tasks in the user program behave. In addition to memory references, packets are written into the trace file when certain events occur, such as creation of a new task or a virtual processor becoming idle. These events can be used to generate parallelism profiles and can be fed into a post-mortem program visualization tool such as ParVis [4]. Tmul-T also generates records of

synchronization events, which can be used for coupled post-mortem simulations. This feature of Tmul-T provides a profiling mechanism similar to the one developed by Davis and Hennessy, who incorporated tracing facilities into high-level synchronization macros [8].

3.4 Memory Allocation

Tmul-T differs from Mul-T in its assumptions about memory. Mul-T assumes a uniform access time memory model. In order to use a Tmul-T trace to simulate an architecture that can take advantage of communication locality, Tmul-T distributes the shared-memory among the virtual processors. The first few records of a Tmul-T trace file include a memory map that specifies the boundaries of three areas of shared memory:

1. The static read-only area contains code and any other objects in the system that are known to be read only.
2. The static writable area contains all other static data.
3. The heap area contains memory objects that the application program allocates dynamically.

The heap is subdivided into equal regions, one for each virtual processor. In the current implementation, each virtual processor allocates memory only in its own region. The Tmul-T environment ignores the difficulties of remote data allocation and garbage collection. We are currently investigating these issues, but the subject of memory management in a shared memory multiprocessor with non-uniform memory latency is difficult and largely unexplored.

3.5 Performance of Tmul-T

Our initial implementation uses one physical processor. Table 1 shows the execution times for several programs in Mul-T, Tmul-T emulation, and Tmul-T tracing mode. The numbers in parentheses indicate the cost of Tmul-T relative to the uninstrumented Mul-T version. These times are for execution on one virtual processor. The trace numbers for speech were not taken due to a lack of disk space.

The factor of ten to twenty slowdown for emulation mode is caused by the overhead of context switching between the virtual processors and writing the trace data into the memory buffer. The difference between the emulation and trace timings could be eliminated by writing out the memory buffer to disk in the background, as in [9].

The other source of overhead arises from simulating n virtual processors on one physical processor and results in an n-fold decrease in performance over an n-way multiprocessor. The tracing performance can be improved by block scheduling of threads on more real processors. A block of threads can be scheduled on some number of physical processors, and whenever a thread makes a memory reference it can be replaced by another. The scheme must, however, address the issue of synchronizing writes into a single log file, or writing time stamps when using multiple private trace buffers.

	boyer		queens		fib		speech	
Mul-T	20.6		2.7		2.3		103.7	
Tmul-T (emulate)	343.8	(17)	40.9	(15)	48.8	(21)	1085.9	(10)
Tmul-T (trace)	555.3	(27)	63.1	(23)	72.0	(31)	—	

Table 1: Execution time in seconds and slowdown for emulation and tracing (one processor).

4 Multiprocessor Performance Analysis Using Address Traces

Address traces represent the behavior of parallel applications programs and can be used in multiprocessor performance evaluation. We have experimented with several trace-driven simulation techniques that help investigate the performance of multiprocessors with caches and networks. All of our evaluation techniques are capable of measuring performance in terms of processor utilization, which isolates the contribution of a multiprocessor's memory system to the speed of the machine as a whole.

When using address traces to evaluate the performance of multiprocessor memory systems, the tradeoff between the complexity and the accuracy of the measurement techniques gives rise to two kinds of trace-driven schemes: coupled and decoupled.

The simpler *decoupled* techniques simulate the various components of the multiprocessor separately, without emulating the feedback between the processor, the memory system, and the interconnection network. Although such decoupled trace-driven simulations are easy to construct, they may not model the behavior of multiprocessors accurately enough to give clear measurements of memory system performance.

More complex techniques simulate the feedback between multiprocessor components. While such *coupled* trace-driven simulations provide accurate measurements of memory system performance, they require much more time to be constructed, run, and administered than decoupled simulations. However, by substituting an analytical model for the portion of a simulation system that models the interconnection network, we show that it is possible to improve the accuracy of decoupled techniques and to reduce the complexity of the coupled techniques.

4.1 Decoupled Versus Coupled Simulation Techniques

This section describes decoupled and coupled trace-driven simulation methods. A hybrid decoupled method that uses an analytical model for the interconnection network is validated using coupled simulations. The validation takes the form of reconciling the differences between the two methods by analyzing the causes of the discrepancies.

Figure 2 illustrates our simulation strategies. The left side of the figure depicts the completely trace-driven, decoupled simulation technique. This technique uses a trace generation system to produce a record of each processor's requests to memory. The decoupled memory system simulator processes the trace by modeling the effect of each request on the memory system and generates a new trace that consists of the network transactions that are needed to service the

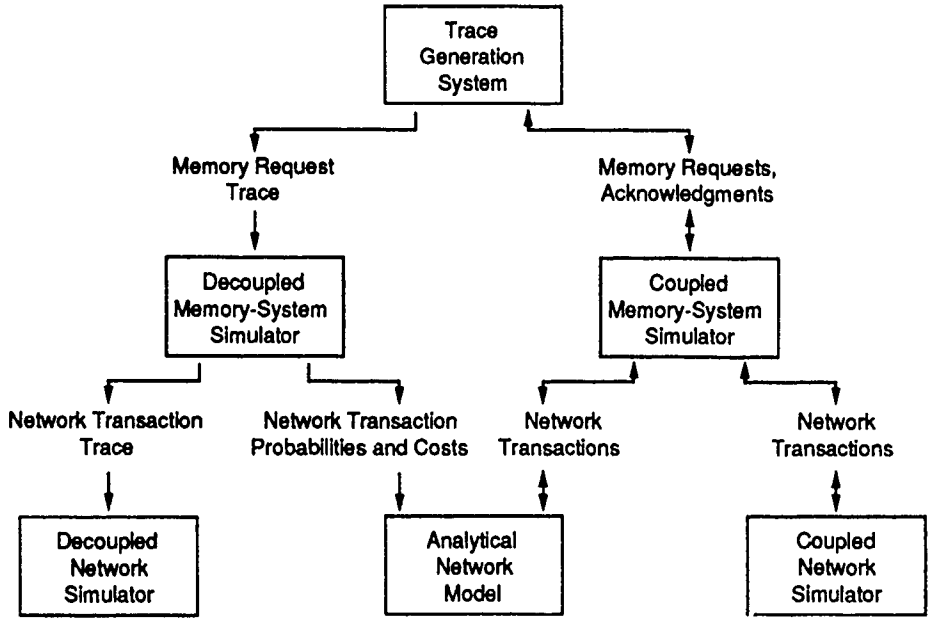

Figure 2: Trace-driven simulation techniques for multiprocessors.

requests. In turn, the decoupled network simulator processes the network transaction trace to determine the processor utilization for the system. In this system, there is no feedback between the components of the evaluation technique.

Without feedback from the memory system to the trace generation system, varying memory access delays cause a skew between the sense of time as determined by the execution of each processor's thread of control. In such a system, each simulated component operates without synchronizing with any other component. Figure 3 shows that the skew caused by the lack of feedback reaches a significant fraction of the total length of a 64-processor machine simulation. The horizontal axis measures the length of the simulation in increments of 2500 cycles, and the vertical axis measures the skew in cycles at each increment on the horizontal axis. Each curve plots the skew between a processor and the slowest processor in the system. Note that the maximum skew between processors reaches twenty percent of the entire length of the simulation!

The time skew in decoupled trace-driven simulations causes two problems. First, since processors do not synchronize properly, the simulation does not represent a correct execution of the traced application. Second, the skew generates huge queues within our event-driven network simulator. Long queue lengths thrash the virtual memory system of the machine used to run the simulations. Thus, not only does the lack of feedback result in experimental error, but it also makes completely trace-driven, decoupled simulation impractical.

A straightforward way to eliminate the time skew problem is to couple the simulation modules with bidirectional interfaces[1]. The right side of Figure 2 depicts this type of system. Each interface in the simulation system mimics the corresponding hardware interface to accurately

[1]A simpler method artificially limits the maximum skew between tasks by temporarily halting all memory requests when the skew reaches a predetermined value. This method gives reasonable results when random transient effects cause skew, but does not work in general.

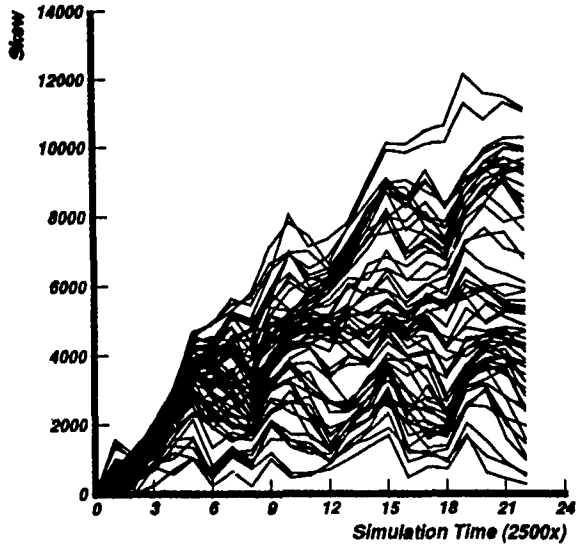

Figure 3: Time skews in a 64 processor machine running the Weather application.

model the feedback in a real multiprocessor. In order to properly simulate the interface between the processor and the memory system, the trace generation system must emit each memory request only after the previous request has completed. A tracing system can use an intermediate address trace file with embedded synchronization information to recreate a correct sequence of memory requests.

Coupled *post-mortem* scheduling systems must also have some means of handling synchronizations. For example, in Kurihara's implementation [17], the system has a choice of spin locks, software combining barrier trees, and adaptive backoff synchronization. The system can also switch among threads on certain exception conditions like remote memory requests, and can simulate a multithreaded processor trace execution.

By accurately modeling all of the interfaces within a multiprocessor, it is possible to closely examine the performance of a memory system. However, a coupled technique requires a much higher investment than decoupled techniques in terms of the time needed for building, running, and administering simulations. Since the overhead is greater for coupled simulations, the technique does not lend itself to rapidly evaluating a number of different memory system implementations.

Figure 2 shows that an analytical model may replace the network simulator in both the coupled and the decoupled simulation techniques. A network model reduces the time investment required by coupled simulations, and preserves the correctness of the decoupled scheme. By calculating communication delay as a function of network load, a model mimics the communication delay for each network transaction without incurring the overhead of simulating every network switch. The network model uses measures of traffic rates, message sizes, and communication locality, to compute communication time (*e.g.* using Equation 1 in Appendix A). While the hybrid methodology neglects localized phenomena within the network, such as hot-spot contention, it correctly models the components of the memory system.

The *hybrid decoupled* strategy combines trace-driven simulations with analytical models that use recursive formulae to compute the effect of the feedback between processors and their memory system. When using the analytical model in the decoupled technique, the memory system simulator processes the entire address trace and calculates the average cost and probability of each possible type of network transaction. These statistics allow a network model to calculate the average processor utilization for the system (*e.g.* using Equation 2 in Appendix A). Using an analytical network model eliminates the skew problem, because the memory system simulation preserves the order of memory accesses in the trace. Although this hybrid decoupled strategy does not accurately model hot-spot contention in the memory system or the network, the next section shows that it successfully predicts the behavior observed in coupled simulations.

4.2 Validating the Hybrid Decoupled Technique

Hybrid evaluation techniques that use a combination of trace-driven simulation and analytical models produce accurate performance measurements of multiprocessor memory systems. Coupled simulations validate the results from the hybrid decoupled technique. We use a trace of a Weather modeling program to discuss the process of reconciling the two methods. A post-mortem scheduler using software combining trees for barrier synchronizations provides the input to both the coupled and decoupled simulators. The simulation systems are similar in that they both implement a number of cache coherence protocols that are designed for large-scale multiprocessors. While the hybrid decoupled technique analytically models a packet-switched multistage network, the coupled technique simulates the same network at the switch level.

In general, we find that the coupled evaluation technique confirms the validity of the estimations of processor utilization by the hybrid decoupled methodology. However, there are differences between the two methodologies in terms of absolute performance measurements that must be justified before trusting the results of the hybrid decoupled simulation technique. This verification process is discussed next.

Figure 4 shows good agreement between the processor utilization results for the Weather application derived from both the coupled and the decoupled simulation techniques for several cache coherence schemes. (See [5], for a description of the coherence schemes.) The match between the different evaluation methodologies requires modifications to the hybrid method to correct two discrepancies. The first discrepancy arises because the two schemes use slightly different basic system parameters. The more detailed functional specification of the cache controller in the coupled simulator adds a few extra cycles to the overhead in network message handling at the source and destination. This simply reflects the fact that it is hard to come up with accurate system parameters without going through a design exercise. Section 4.3 discusses the parameter adjustment that is needed to reconcile the coupled and decoupled simulation results.

Second, a variable in the Weather application causes hot-spot memory accesses, whose effect is not captured by the network model in the hybrid method. Section 4.4 examines the effects of this variable and discusses the performance after optimizations remove the hot-spot behavior.

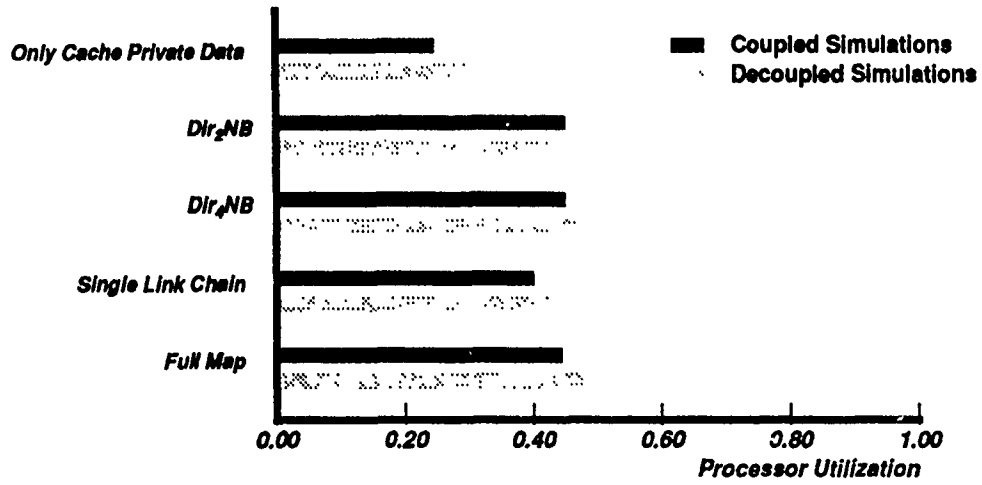

Figure 4: Comparison of processor utilization measurements for the Weather application, obtained from coupled and decoupled evaluation methodologies.

4.3 Reconciling Decoupled and Coupled Simulations

The coupled simulation technique models features of the memory controllers, including finite state machines, network buffers, and internal contention for resources. Since the decoupled methodology does not perform such a detailed simulation, the memory controller runs slower in the coupled simulations than it does in the decoupled technique's network model. Figure 5 shows that the absolute performance measurements from the two simulation strategies do not correlate, due to differences in the way that the controller is modeled.

In order to reconcile the two evaluation techniques, we adjust the model's memory access time to match the behavior of the memory system simulated by the coupled technique. By showing the relationship between base memory access time and processor utilization, Figures 6(a) and (b) extend the predictions of the network model into the range of memory latency observed for the Weather application with full-map and limited directory protocols. The curve on each of the graphs shows the prediction of the network model for a range of memory latencies, given the average request rate and the average block size calculated from the decoupled simulations. The square on each graph shows the prediction of the model for the memory latency assumed in the decoupled technique. Since this point is *calculated* from the network model, it sits on the prediction curve. The triangles label the *observed* processor utilizations and average memory latencies in coupled simulations of the Weather application.

In the coupled simulations, different latencies can be created by changing parameters such as the time needed to modify a directory entry or the structure of the finite state machines within each memory controller. The reported latency values are calculated by subtracting twice the average network latency from the average total access latency of remote memory transactions. Thus, the reported memory latency values include all of the delay needed to service a transaction (including invalidations), except for the time needed to transport protocol messages through the network.

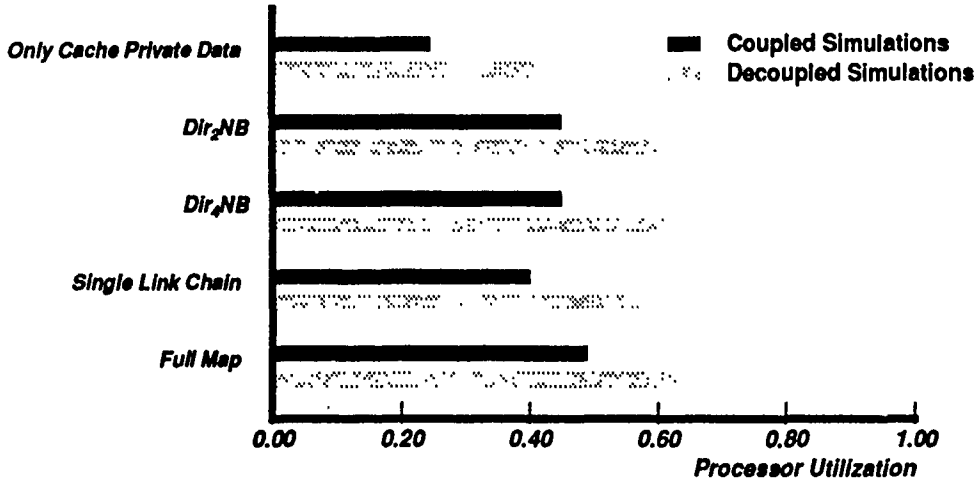

Figure 5: Comparison of processor utilization measurements for Weather, before adjusting the base memory access latency.

When the base memory access time used in the decoupled technique is adjusted to correspond to the memory latency observed by coupled simulation, the different analysis methods yield similar processor utilization measurements. Figure 7 compares the results from the two techniques with the adjusted network model. We see that only the predictions for the single-link chain and full-map schemes are close.

4.4 An Advantage of Coupled Simulations

Although adjusting the memory access time corrects for the absolute difference between the predicted and observed processor utilizations, Figure 7 shows that the adjustment does not completely reconcile the results of the coupled and decoupled simulation techniques. Specifically, the decoupled simulations predict that the limited directories perform almost as well as the full-map directory, but the coupled simulations demonstrate that the limited directories realize lower processor utilizations than the full-map protocol.

The discrepancy between the predicted and the actual performance of limited directory protocols in the decoupled technique results from averaging the effects of the data requests over the entire duration of a trace and over all of the components in the simulated multiprocessor. This methodology does not account for hot-spot contention, which results from a concentration of requests impinging on a single component.

The Weather application uses a variable that belongs to the class of *write-once* data. Since the variable is read frequently by all of the processors in the system after being written once, the limited directory protocol produces a constant flow of data requests from every processor in the system to the memory module that contains the variable. While the decoupled methodology averages this hot-spot traffic over the entire multiprocessor, the coupled simulation captures the hot-spot effect.

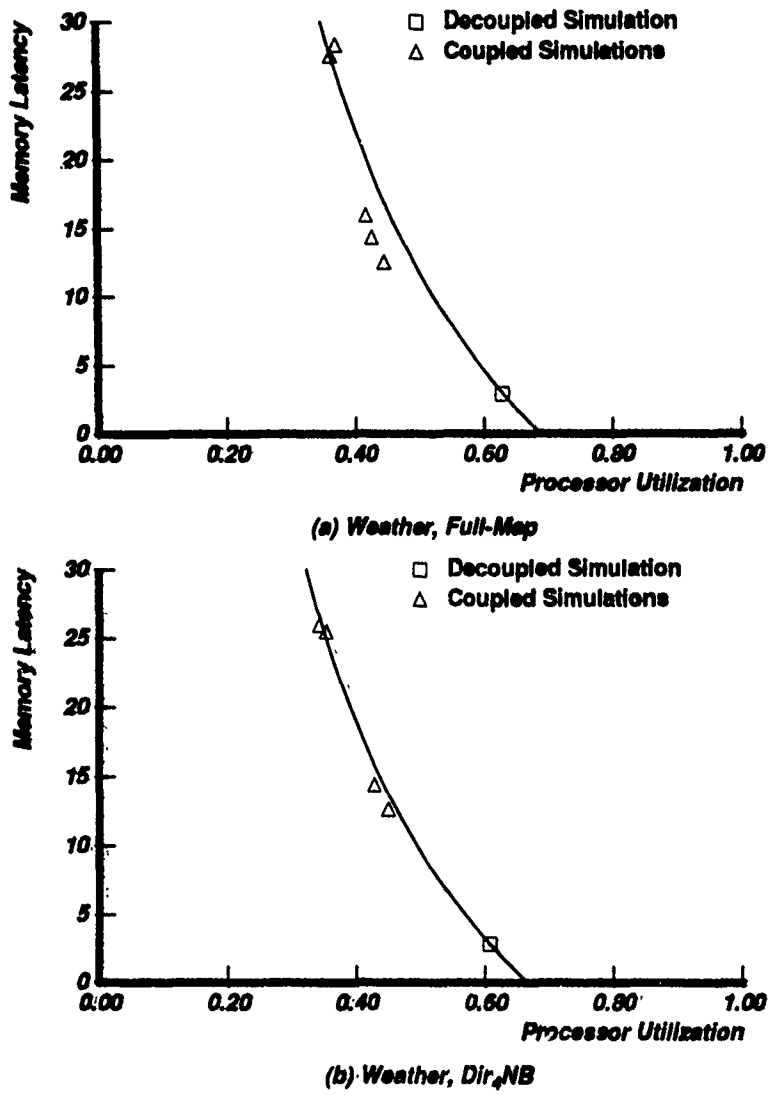

Figure 6: Processor utilization versus memory latency. The curve indicates the prediction of the network model. The individual points are data from simulations.

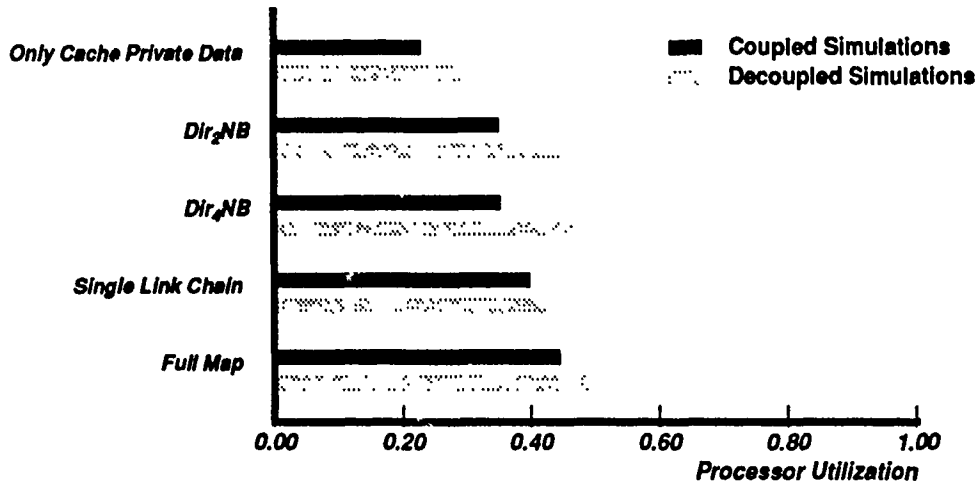

Figure 7: Comparison of processor utilization measurements for Weather, after adjusting memory access latency, but before eliminating the hot-spot.

The coupled simulation system implements features that allow the elimination of this hot-spot. We modified the dynamic post-mortem scheduler to mark the write-once data location with a special memory access code. The memory system protocol interprets the special code and prevents the data access from reserving pointers in the limited directory data structure, thereby allowing an unlimited number of cached copies of the write-once data location. By avoiding the flow of data requests caused by limited directory evictions, this data access mechanism eliminates the hot-spot problem.

Figure 8 verifies the solution of the hot-spot caused by Weather's write-once variable. The graph shows a histogram of the size of the cache controller network queues for coupled simulations with and without hot-spot contention. Since network queues store protocol messages that memory modules need to transmit through the network, the histogram indicates the amount of time that memory requests have to wait to be serviced. The solid curve on the histogram shows the behavior of the system with the hot-spot data accesses, and the dashed curve shows the performance once the write-once variable has been optimized. Figure 8 illustrates the fact that hot-spot contention causes thousands of protocol messages to wait in long queues. However, using the mechanism described above effectively removes the hot-spot.

After the hot-spot has been removed, the processor utilizations observed for the limited directory schemes conform to the prediction of the decoupled simulation technique. Figure 4 shows the processor utilizations of the Weather program after the hot-spot has been resolved. Although the results from the two simulation techniques correlate well, the performance of the protocols in coupled simulations remains slightly below the predictions of the decoupled methodology, due to the non-uniform distribution of requests to memory modules. Nevertheless, the coupled trace-driven simulations validate the fundamental conclusions derived from the hybrid decoupled technique.

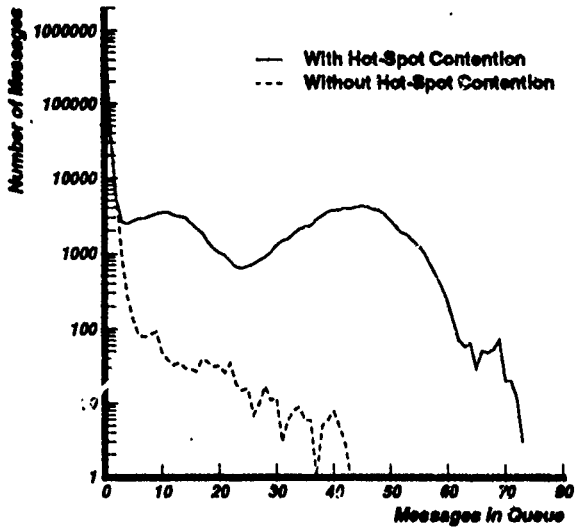

Figure 8: Cache controller queue sizes with Dir_4NB protocol.

5 Conclusions and Future Work

This paper compared several methods for multiprocessor performance evaluation using address traces. We describe a compiler-aided tracing scheme called Tmul-T that traces an arbitrary number of processors with less than a factor of 20 slowdown when tracing one processor. We also validate a trace-driven simulation method that simulates the caching system and uses an analytical model for the interconnection network by comparing its predictions to simulations that couple the caches and the network.

Multiprocessor performance evaluation techniques based on address traces can be largely classified as decoupled and coupled. Decoupled techniques simulate the various components of the multiprocessor separately, while coupled techniques simulate the feedback between multiprocessor components. Although decoupled trace-driven simulations are easy to construct, they may not model the behavior of multiprocessors accurately enough to give clear measurements of memory system performance. Coupled trace-driven simulations provide accurate measurements of memory system performance, but they require much more time to be constructed, run, and administered than decoupled simulations.

However, by substituting an analytical model for the portion of a simulation system that models the interconnection network, it is possible to improve the accuracy of decoupled techniques and to reduce the complexity of the coupled techniques. We describe a hybrid decoupled simulation scheme that simulates the cache and memory subsystem, but uses an analytical model for the interconnection network. We validate this method by comparing its accuracy with coupled simulations, and show that even though the hybrid decoupled scheme is over an order of magnitude faster than a coupled scheme, its accuracy is comparable to the coupled scheme. We also found that coupled trace-driven simulations do not have a speed advantage over full system simulations, but such simulators are much easier to write than a full system simulator.

Given these observations, how should computer designers choose a simulation technology?

The answer is best illustrated using the Alewife evaluation methodology as an example. At various stages of the design process, different technologies are suitable. In the initial design phase, performance numbers over a wide range of parameters are needed, so analytical modeling and hybrid decoupled methods are most appropriate. In the Alewife effort, several cache coherence schemes and network types were rapidly evaluated using models and hybrid decoupled schemes.

In the next phase of the design process, full system simulations yield more detailed results that establish the features that are ultimately bound into hardware. Although coupled trace-driven simulations validate the conclusions from the hybrid decoupled analysis and provide some insight into transient behavior, our experience has shown that they are not integral to the design process. There are several reasons for this conclusion. First, our coupled simulator is not significantly faster than ASIM, our full system simulator. Second, we find that the hybrid decoupled scheme is significantly faster than the coupled scheme, and its accuracy is comparable. Third, our coupled simulator is less flexible than ASIM, because it allows only limited modifications to the processor, the compiler, and the application. However, coupled trace-driven analysis benefits projects that focus on investigating specific aspects of memory system design. In such projects, the investment of time required to write a detailed full system simulator may not be justified.

6 Acknowledgments

Pat Teller and Allan Gottlieb of New York University helped us obtain the source code for the Weather application, and Harold Stone at IBM helped us obtain the trace. Kiyoshi Kurihara found the write-once variable in the Weather application. Gino Maa wrote the network simulator that was used for running the coupled simulations and for validating our network model. The research reported in this paper is funded by DARPA contract # N00014-87-K-0825 and by grants from the Sloan F\ ndation and IBM.

References

[1] Hal Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Anant Agarwal. Limits on Network Performance. November 1989. Laboratory for Computer Science, M.I.T. MIT VLSI Memo 1989. Submitted for publication.

[3] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 119–127, IEEE, New York, June 1986.

[4] Laura Bagnall. *ParVis: A Program Visualization Tool for Multilisp.* Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1989.

[5] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, June 1990.

[6] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. August 1990. Laboratory for Computer Science, M.I.T. MIT VLSI Memo. Submitted for publication.

[7] Mathews Cherian. *A Study of Backoff Barrier Synchronization in Shared-Memory Multiprocessors.* Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[8] Helen Davis and John Hennessy. Characterizing the synchronization behavior of parallel programs. In *Proceedings of the ACM/SIGPLAN PPEALS Parallel Programming: Experience with Applications, Languages and Systems*, ACM, July 1988. Published as Vol. 23, No. 9, of SIGPLAN Notices.

[9] Susan J. Eggers, David R. Keppel, Eric K. Koldinger, and Henry M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *Proceedings of ACM SIGMETRICS 1990*, May 1990.

[10] Robert H. Halstead. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[11] Robert R. Henry. Tracer - Address and Instruction Tracing for the VAX Architecture. University of California, Berkeley, November, 1984.

[12] Mark Holliday. Trace-Driven Simulation of Distributed Shared Memory Environments. June 1990. The 17th Annual International Symposium on Computer Architecture, Workshop III: Scalable Shared-Memory Architectures.

[13] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implemenation*, June 1989.

[14] D. Kranz et al. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of SIGPLAN '86, Symposium on Compiler Construction*, June 1986.

[15] Clyde P. Kruskal and Marc Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, December 1983.

[16] Manoj Kumar and Kimming So. Trace Driven Simulation for Studying MIMD Parallel Computers. In *International Conference on Parallel Computing*, pages I-68 – I-72, 1989.

[17] Kiyoshi Kurihara. *Performance Evaluation of Large-Scale Multiprocessors.* Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1[illegible].

[18] J. Rees, N. Adams, and J. Meehan. *The T Manual, Fourth Edition.* Technical Report, Yale University, Computer Science Department, January 1984.

[19] Arturo Salz. VTRACE. 1984. Computer Systems Laboratory, Stanford University.

[20] Richard L. Sites and Anant Agarwal. Multiprocessor Cache Analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 186–195, IEEE, New York, June 1988.

[21] K. So, F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. *PSIMUL - A System for Parallel Simulation of Parallel Systems.* Technical Report RC 11674 (58502), IBM T. J. Watson Research Center, Yorktown Heights, November 1987.

[22] Craig B. Stunkel and W. Kent Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. In *Proceedings of ACM SIGMETRICS 1989*, May 1989.

A Performance Models for Interconnection Networks

Hybrid, decoupled trace-driven simulation schemes can use the following network models to estimate network latency. We first present a simple packet-switched, buffered, multistage interconnection network model. The network switches (of size $k \times k$) are pipelined so a message header can leave a switch even while the rest of the message is still being serviced. A network request travels through n switch stages to the destination node and takes M cycles for the memory access. The response to the request (or the acknowledgment in some cases) is inserted into the network by the destination node. Let us further assume that the processor idles from the time that its network request is serviced until it receives a response.

Computation of the processor utilization with multistage networks is based on the network model proposed by Kruskal and Snir [15]. Using the usual assumptions of independent requests uniformly distributed over all the memory modules, the network model yields the average latency T of a memory request through the network. In the equation below, ρ, the channel utilization, is the product of the effective network request rate m and the average message size B.

$$T = \left[1 + \frac{\rho B\left(1 - \frac{1}{k}\right)}{2(1-\rho)}\right] n + B + M - 1 \tag{1}$$

The latency is n times the delay through a switch (note that acknowledgments are included in the request rate), plus the memory delay and the message pipeline delay. The switch delay is one plus contention delay. We can now compute processor utilization U using the following set of equations:

$$\begin{aligned} U &= \frac{1}{1+mT} \\ \rho &= UmB \\ T &= \left(1 + \frac{\rho B\left(1 - \frac{1}{k}\right)}{2(1-\rho)}\right) n + B + M - 1 \end{aligned}$$

The above equations can be solved using standard numerical methods. A closed form solution for U can also be derived as:

$$U = \frac{1}{1 + \frac{m}{2}\left[n + M + 2B - 1 - \frac{1}{m} + \sqrt{(n + M - 1 + \frac{1}{m})^2 + 2nB^2\left(1 - \frac{1}{k}\right)}\right]} \tag{2}$$

A k-ary n-cube direct network can be modeled in a similar fashion [2] by replacing the expression for T in Equation 1 with,

$$T = \left[1 + \frac{\rho B \frac{1}{k_d}\left(1 - \frac{1}{k_d}\right)\left(1 + \frac{1}{n}\right)}{(1-\rho)}\right] n k_d + B + M - 1 \qquad (3)$$

where k is the network radix, n is the dimension, and k_d is the average distance a message travels in a dimension. For a torus with channels in both directions, $k_d = k/4$.

B Description of Tmul-T Trace File Format

The trace file is a sequence of 6-byte packets. The first byte is the processor number and the second byte is an opcode to be described below. The next four bytes are generally the longword address but this depends on the particular opcode. In writing programs to filter the trace note that packets follow the VAX byte order (they are little-endian). The first seven opcodes described in the table are always in the first seven packets. Opcode descriptions:

32	The address is the beginning of the static read-only area. (.text)
33	The address is the end of the static read-only area.
34	The address is the beginning of the static writable area. (.data)
35	The address is the end of the static writable area.
36	The address is the beginning of the heap.
37	The address is the end of the heap. Note that the heap is equally divided among the processors from 1 to n.
38	The address is the number of processors, n.
0	The address was read.
1	The address was written.
2	The address was read and written, e.g. add to memory.
3	The address was test and set. (synchronization)
8	The address was read from the stack.
9	The address was written to the stack.
10	The address was read and written to the stack.
16	The task blocked on a future.
17	A task that blocked was restarted.
18	A new future started running.
19	The processor became idle (looking for work).
20	A task started determining its future.
21	A task finished determining its future.
22	A future started to be created.
23	The creation of the future has completed.

Any address outside the ranges indicated by codes 32 through 37 belongs to an object created when the program was loaded or initialized. Codes 16 through 23 indicate actions by the scheduler regarding task scheduling. The address field of the packet is ignored for these codes.

OFFICIAL DISTRIBUTION LIST

DIRECTOR Information Processing Techniques Office Defense Advanced Research Projects Agency (DARPA) 1400 Wilson Boulevard Arlington, VA 22209	2 copies
OFFICE OF NAVAL RESEARCH 800 North Quincy Street Arlington, VA 22217 Attn: Dr. Gary Koop, Code 433	2 copies
DIRECTOR, CODE 2627 Naval Research Laboratory Washington, DC 20375	6 copies
DEFENSE TECHNICAL INFORMATION CENTER Cameron Station Alexandria, VA 22314	12 copies
NATIONAL SCIENCE FOUNDATION Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
HEAD, CODE 38 Research Department Naval Weapons Center China Lake, CA 93555	1 copy